Comparing Proof Techniques on the Simply Typed Lambda Calculus between ITT and HoTT

Yanjun Yang, 2020

Advisor: David Walker

March 26, 2019

1 Extrinsically Typed $\lambda^{\rightarrow}$ with de Brujin Indices in ITT

First, we worked with an extrinsically typed lambda calculus. In an extrinsically typed lambda calculus, the type of the lambda term exists separately from the lambda term itself. In this language, there are two types: a base type Boolean which contains normal forms are True and False, and a type-constructor $t \rightarrow t'$ whose normal forms are all well-typed lambda abstractions. There are additionally two other terms in the language, function application and variables, whose names are de Brujin indices. Contexts in this language are an ordered list of types of terms. The following is the language in Backus normal form.

$$\tau ::= \tau \rightarrow \tau \mid \text{bool}$$

$$\Gamma ::= \varnothing \mid \Gamma, \tau$$

$$i_\Gamma ::= (n, \tau), \text{where } n \in [|\Gamma|] \text{ and } \Gamma[n] = \tau$$

$$e ::= \text{true} \mid \text{false} \mid i_\Gamma \mid \lambda\tau.e \mid e\ e$$

$$v ::= \text{true} \mid \text{false} \mid \lambda\tau.e$$

The following are the typing rules of the language:

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}\ \text{T-True} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}\ \text{T-False} \quad \frac{n \in [|\Gamma|] \qquad \Gamma[n] = \tau}{\Gamma \vdash (n, \tau) : \tau}\ \text{T-Var}$$

$$\frac{\Gamma, \tau \vdash e : \tau'}{\Gamma \vdash \lambda\tau.e : \tau \to \tau'}\ \text{T-Abs} \quad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ \text{T-App}$$

The language uses call-by-value evaluation order. We use the notation $e \longrightarrow e'$ to denote that e steps to e' by using one of the evaluation rules. The following are the evaluation rules of the language

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{E-App1} \quad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{E-App2} \quad \frac{\Gamma \vdash \lambda\tau.e : \tau \to \tau' \qquad \Gamma \vdash v : \tau}{(\lambda\tau.e)\ v \longrightarrow [v/(0, \tau)]e}\ \text{E-AppAbs}$$

Note that the rule E-AppAbs requires that both the substitutand and substitute to be well-typed. With these definitions, we can now prove properties about the language.

Theorem 1. *(Progress) For all expressions e and types τ, if $\varnothing \vdash e : \tau$, then either e is a value or there exists an e' such that $e \longrightarrow e'$*

Proof. By induction on the structure of the typing derivation $\varnothing \vdash e : \tau$. □

Lemma 2. *(Uniqueness of Types) For all contexts Γ, expressions e and types τ, τ', if $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$, then $\tau = \tau'$.*

Proof. By induction on the structure of the expression e. □

Definition 3. *For all contexts Γ and variables i_Γ, let $\Gamma - i_\Gamma$ be the resulting context where i_Γ is removed from Γ.*

Lemma 4. *(Weakening) For all contexts Γ, variables i_Γ, expressions e and types τ, if $\Gamma - i_\Gamma \vdash e : \tau$, then $\Gamma \vdash e : \tau$.*

Proof. By induction on the structure of $\Gamma - i_\Gamma \vdash e : \tau$ □

Lemma 5. *(Preservation of Types under Substitution) For all contexts* Γ*, variables* i_Γ*, expressions* e_1, e_2*, and types* τ_1, τ_2*, if* $\Gamma \vdash i_\Gamma : \tau_1$*,* $\Gamma \vdash e_1 : \tau_2$*, and* $\Gamma - i_\Gamma \vdash e_2 : \tau_1$*, then* $\Gamma - i_\Gamma \vdash [e_2/i_\Gamma]e_1 : \tau_2$*.*

Proof. By induction on the structure of $\Gamma \vdash e_1 : \tau_2$, using the Weakening lemma for substituting into a function body. □

Theorem 6. *(Preservation) For all expressions* e, e' *and all types* τ*, if* $\varnothing \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\varnothing \vdash e' : \tau$*.*

Proof. By induction on the structures of $\varnothing \vdash e : \tau$ and $e \longrightarrow e'$, using the Uniqueness of Types and Preservation of Types under Substitution lemmas for the E-AppAbs case. □

2 Intrinsically Typed $\lambda^\to$ with de Brujin Indices in ITT

Next, we attempted the same proof, but using an intrinsically typed $\lambda^\to$. The key difference is that terms in the language are always tagged with their types. The following is the new BNF of the language:

$$\tau ::= \tau \to \tau \mid \text{bool}$$

$$\Gamma ::= \varnothing \mid \Gamma, \tau$$

$$i_\Gamma^\tau ::= (n, \tau), \text{where } n \in [|\Gamma|] \text{ and } \Gamma[n] = \tau$$

$$e^\tau ::= \text{true}^{\text{bool}} \mid \text{false}^{\text{bool}} \mid i_\Gamma^\tau \mid (\lambda\tau.e^\tau)^\tau \mid (e^\tau e^\tau)^\tau$$

$$v ::= \text{true}^{\text{bool}} \mid \text{false}^{\text{bool}} \mid (\lambda\tau.e^\tau)^\tau$$

However, there are constraints on what the types of the expressions can be. Namely:

$$\frac{\tau \quad e^{\tau'}}{(\lambda\tau.e^{\tau'})^{\tau\to\tau'}} \qquad \frac{e^{\tau\to\tau'} \quad e^\tau}{(e^{\tau\to\tau'}\ e^\tau)^{\tau'}}$$

With these constraints, typing an expression in this language becomes trivial: $e^\tau : \tau$. The execution rules are analogous to the ones for the extrinsically typed language, with the added constraint that expressions always evaluate to another expression of the same type. The proof of the Progress theorem is analogous to the extrinsically-typed case, but the proof of the Preservation theorem in the intrinsically typed case becomes trivial given how it is stated:

Theorem 7. *(Preservation for intrinsically typed $\lambda^\to$) For all types τ and all expressions e^τ, e'^τ, if $e^\tau : \tau$ and $e^\tau \longrightarrow e'^\tau$, then $e'^\tau : \tau$.*

Proof. By definition, $e'^\tau : \tau$. □

3 Extrinsically Typed $\lambda^\to$ with Names in ITT

We now attempt the same proofs again, but this on a $\lambda^\to$ with named variables. The difference between this version of the $\lambda^\to$ and the ones above is that variable names, unlike de Brujin indices, need not be unique. Therefore, when a new variable that is introduced has the same name as another existing variable, the previous variable is shadowed and is no longer accessible. The following is the BNF for this language:

$$\tau ::= \tau \to \tau \mid \text{bool}$$

$$\Gamma ::= \varnothing \mid \Gamma, x : \tau$$

$$e ::= \text{true} \mid \text{false} \mid x \mid \lambda x : \tau . e \mid e\ e$$

$$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau . e$$

The typing rules are the same as the above extrinsically-typed example, with the following modifications to reflect the fact that we are using named variables:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}\ \text{T-Abs}$$

Note that $x : \tau \in \Gamma$ means that $x : \tau$ is the most recent binding of the name x with any type in Γ (i.e. if $\Gamma = \varnothing, x : \tau', x : \tau$, then $x : \tau \in \Gamma$, but $x : \tau' \notin \Gamma$).

By convention, substitution must be capture-avoiding, which means a substitution into the body of a lambda abstraction can only occur if the variable that is bound by that lambda abstraction is not in the free variables of the substituting expression (i.e. $[e_2/x](\lambda y : \tau.e_1) = \lambda y : \tau.([e_2/x]e_1)$ if and only if $y \notin FV(e_2)$). Consequently, if a lambda abstraction binds a variable with the same name as the substituted variable, then the body of the lambda abstraction is unaffected(i.e. $[e_2/x](\lambda x : \tau.e_1) = \lambda x : \tau.e_1$), since the substituted variable is shadowed inside the body. However, since our language uses call-by-value execution order, we know that when we perform a substitution, the substituting term is always closed (i.e. it has no free variables). Therefore, we only need to check for variable shadowing when performing capture-avoiding substitution.

The proofs of type safety are the same except that they must be modified to accommodate the new definition of substitution. The proof of Progress is completely analogous, but the proof of Preservation requires additional lemmas:

Definition 8. *(Sublist) For all contexts Γ, Δ, Γ is a sublist of Δ (written as $\Gamma \subseteq \Delta$) if and only if they satisfy the following inductive definition:*

1. *For all contexts Δ, $\varnothing \subseteq \Delta$.*

2. *For all contexts Γ, Δ, if $\Gamma \subseteq \Delta$, then for all variables x and types τ, $\Gamma, x : \tau \subseteq \Delta, x : \tau$.*

Lemma 9. *(Sublist Typing) For all contexts Γ, Δ, expressions e and types τ, if $\Gamma \vdash e : \tau$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash e : \tau$.*

Proof. By induction on the structure of $\Gamma \vdash e : \tau$. □

Lemma 10. *(Shadowing) For all context Γ, natural numbers i, j, expressions e and types τ, if $\Gamma \vdash e : \tau$ and the i-th variable of Γ shadows the j-th variable of Γ, then $\Gamma - j \vdash e : \tau$.*

Proof. By induction on the structure of $\Gamma \vdash e : \tau$. □

The proof of Preservation of Types under Substitution will follow the same structure as before, except that we will use the Sublist Typing lemma on the substituting expression (which is closed and therefore typed in context $\varnothing$) when replacing the substituted variable, and we will use the Shadowing lemma when substituting into a lambda abstraction that shadows the substituted variable. The proof of Preservation is unchanged.